VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Využití systému ECJ pro Evoluční výpočetní techniky

Using ECJ system for Evolutionary Computation Research

2015

Michal Koudela

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Michal Koudela**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Využití systému ECJ pro Evoluční výpočetní techniky
Using ECJ system for Evolutionary Computation Research

Zásady pro vypracování:

Práce bude zaměřena na využití A Java-based Evolutionary Computation Research System (dále jen ECJ), který byl vyvinut pro Evoluční výpočetní techniky na Java platformě.

Cíl práce:
1. Popis technik Genetických algoritmů a Genetického programování, jejich využití a výhody, srovnání obou techik.
2. Analýza ECJ systému, dílčích evolučních technik, které systém poskytuje, především Genetického programování. Srovnání s dalšími systémy zabývající se obdobnou problematikou na různých platformách.
3. Implemetace vlastní aplikace využívající Genetického programování z ECJ systému. Aplikace bude zaměřena na vývoj umělé inteligence v interaktivní hře s grafickým uživatelským rozhranním.
4. Dokumentace a popis struktury aplikace.

Seznam doporučené odborné literatury:

[1] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, Frank D. Francone : Genetic programming: an introduction: on the automatic evolution of computer programs and its applications, Morgan Kaufmann Publishers Inc. San Francisco CA USA, ISBN: 1-55860-510-X
[2] ECJ 20 : A Java-based Evolutionary Computation Research System[online], 2010 [cit. 29-9-2013], George Mason University's ECLab Evolutionary Computation Laboratory, dostupné z WWW: <http://www.cs.gmu.edu/~eclab/projects/ecj>

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Tomáš Buriánek**

Datum zadání: 01.09.2014

Datum odevzdání: 07.05.2015

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky OSTRAVA

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této bakalářské/diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských/magisterských programech VŠB-TU Ostrava.

V Ostravě 7. května 2015 Koudela

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7.května 2015 Koudela

Rád bych na tomto místě poděkoval Ing. Tomáši Buriánkovi za odborné vedení během mé práce.

Abstrakt

V úvodní části se práce zaměřuje na problematiku Genetických algoritmů a Genetického programování a jejich podrobný popis. Dále pokračuje s popisem frameworku ECJ napsaného v jazyce JAVA a jeho způsob implementace evolučních technik. Poslední část této práce se zaměřuje na praktickou část užití evolučních technik systému ECJ pro tvorbu algoritmu ovládající umělou inteligenci v jednoduché hře.

Klíčová slova: ECJ,JAVA,umělá inteligence

Abstract

First part of this thesis focuses on description and analysis of Genetic algorithm and Genetic programming. Following section is focused on detailed description of JAVA framework ECJ used for evolution computing. Final section of this thesis is practical part focusing on implementation of simple game of Snake using ECJ to calculate algorithm that controls artificial intelligence which controls the movement of Snake.

Keywords: artificial intelligence,thesis,framework

Seznam použitých zkratek a symbolů

GP	–	Genetické programování
ECJ	–	Java-based Evolutionary Computation Research System
GA	–	Genetický algoritmus
UI	–	Umělá inteligence

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Tato práce se zabývá problematikou Genetického programování a Genetických algoritmů, jejich využití, rozdílu, výhod a nevýhod.

V první části práce bude stručně popsána funkce ,využití ,porovnání a výhody Genetického programování a Genetických algoritmu.

Druhá část této práce bude zaměřena na důkladnou analýzu Frameworku ECJ – Evolution Computing for Java napsaného v jazyce JAVA, popis dílčích evolučních technik, které poskytuje a dalších doplňkových funkcí. Primárně se práce v této části podrobně zaměří na Genetické programování, kterého je rovněž využito v aplikaci představené níže. Závěrem se tato část bude zabývat systémy, zabývajícími se Evolučními výpočty implementovanými na jiných platformách a jejích porovnání se systémem ECJ.

Třetí část je zaměřená na dokumentaci a popis vlastní aplikace představující interaktivní hru Hada, obsahující umělou inteligenci vytvořenou pomocí evoluční techniky Genetického programování poskytující systém ECJ.

2 Evoluční výpočetní techniky

Tato část práce se zabývá podrobným popisem evolučních technik genetického programování a genetického algoritmu. V každé části je popsána reprezentace dat, průběh evolučního výpočtu a forma výstupu dat tohoto výpočtu.

2.1 Genetický algoritmus

Genetický algoritmus je způsob hledání řešení na komplexní problémy, na které neexistuje exaktní algoritmus. K tomuto hledání používá evoluční techniky napodobující evoluční procesy známé z biologie. [4]

2.1.1 Reprezentace dat v Genetickém algoritmu

Na rozdíl od Genetického programování jsou data v Genetickém algoritmu většinou prezentována skupinou bitů reprezentující daného jedince. Tento formát se ovšem může i lišit a reprezentace je možná například pomocí hodnotového typu Integer nebo speciálních typů kombinující tyto způsoby reprezentace. [4]

2.1.2 Průběh Genetického algoritmu

Průběh Genetického algoritmů je obdobný jako průběh Genetického algoritmu využitého v Genetickém programování. Rozdíl nastává v reprezentaci a tvorbě jedinců reprezentujících potencionální řešení problému a evolučních technik. Například v případě genetického programování, kdy jsou daní jedinci reprezentování formou stromu, technika *Mutace* může v rámci stromu přehodit nebo změnit hodnoty daných uzlů, u Genetického algoritmu jsou jedinci prezentováni skupinou binárních čísel a funkce mutace mění např jen hodnotu jednoho bitu. [4]

1. Inicializace - vytvoření populace.
2. Výběr jedinců z dané populace s vysokým Fitness ohodnocením.
3. Použitím evolučních metod uvedených níže, vygeneruje z vybraných jedinců jedince nové Generace.
   - Mutace
   - Křížení
4. Výpočet Fitness ohodnocení pro jedince nově vzniklé generace.
5. Otestování jedinců v případě nalezení ideálního jedince, je algoritmus ukončen, jinak pokračuje od bodu 2.
6. Ukončení a zobrazení výstupu algoritmu, v tomto případě jedinec s největším Fitness ohodnocením.

Vytvoření prvotní generace
Prvním krokem Genetického algoritmu je vytvoření nulté generace, která se skládá z náhodně vygenerovaných jedinců. Velikost této populace se odvíjí od povahy řešeného problému, ale danou populaci můžou tvořit až stovky jedinců, taktéž velikost daného jedince se odvíjí od charakteru řešeného problému. [4]

Výběr jedinců pro výpočet
Výběr jedinců pro evoluční výpočet probíhá náhodně. Jedinci s vyšším ohodnocením Fitness mají přirozeně větší šanci na výběr.[4]

Vytvoření nové generace pomocí evolučních technik
Ze skupiny vytvořené v předešlém kroku se vyberou dva jedinci popř. rodiče, nad kterými se vykonají operace *Mutace* nebo *Křížení*. Tyto operace se nemusí vždy vykonat, a obě mají určitou míru pravděpodobnosti na vykonáni. Po provedení těchto operací se vytvoří nový jedinec, nesoucí vlastnosti svých rodičů. [4]

- **Mutace** V případě mutace má každý gen pravděpodobnost s kterou bude zmutován. Mutace se provede změnou daného bitu, např. převedení hodnoty 0 na 1 a obráceně.
- **Křížení** Kříženi je technika, která vzájemně zamění určité části jedinců mezi sebou. Např. hodnota určitého bitu z jedince A se přenese na hodnotu tohoto bitu v jedinci B.

Přiřazení ohodnocení Fitness nové generaci
Po vytvoření nové generace, se každý jedinec dané generace ohodnotí funkcí přiřazující Fitness. Toho ohodnocení každého jedince označuje jeho kvalitu řešení určeného problému. Pokud je v dané generaci přítomen jedinec s ideálním Fitness ohodnocením, algoritmus je ukončen a tento jedinec prezentován na výstupu algoritmu. V opačném případě algoritmus nadále pokračuje od kroku 2 ,dokud nenalezne ideálního jedince nebo dokud není dosažen maximální počet generací. [4]

Elitismus
Jeden z problému v Genetickém programování a genetickém algoritmu je v případě ,kdy se v dané generaci nachází jedinci s velkým Fitness ohodnocením, ale nebyli vybrání do tvorby další generace, v tomto případě by byli daní jedinci ztraceni, a kvalita genů v nové generaci by se mohla zhoršit, oproti původní generaci. Tento problém je řešen pomocí Elitismu, kdy jsou nejvýkonější jedinci dané generace přesunutí beze změny do nové generace a je tím zamezená možnost, že by nastala situace, kdy by kvalita řešení nové generace byla horší než kvalita řešení stávající generace.[4]

Nevýhody Genetického algoritmů
Hlavní nevýhodou Genetických algoritmů je jejich špatná aplikovatelnost na komplexní problémy. V případě náročnějších výpočtu, kde se daní jedinci skládají ze stovek bitu.

Funkce ohodnocující dané jedince a přiřazující Fitness ohodnocení, se v takovém případě stávají komplexní a vyžadují několik hodin až dní pro plnou simulaci problému. V takovém případě musí být problém rozdělen do menších částí, které se vyhodnocují samostatně. [4]

2.2 Genetické programování

Genetické programování, nadále jen GP, je metodologie hledání počítačových programů využívající Genetických algoritmů pro řešení problému zadaného člověkem. Je to také způsob strojového učení sloužící k optimalizaci počítačových programů. Je to specializace genetických algoritmů, kdy je každý jedinec prezentován formou počítačového programu. Program vytvořený tímto způsobem, je v zásadě sada instrukcí, uspořádaná určitým způsobem např. ve formě stromu, kterému je pomocí Fitness přiděleno určité ohodnocení. Fitness funkce přiřazuje ohodnocení výkonnosti každému jedinci dané populace na základě jeho výkonnosti. Tato výkonnost se odvíjí od schopnosti řešit problém stanovený daným uživatelem. Pravidla pro vyhodnocování této výkonnosti stanovuje uživatel ,v tomto případě programátor. [2][3]

2.2.1 Reprezentace dat v Genetickém programování

V Genetickém programování se data mohou prezentovat dvěma různými způsoby, pomocí stromové struktury nebo v případě Lineárního genetického programování, jsou data prezentována formou po sobě jdoucích instrukcí nebo příkazů.

U tohoto způsobu reprezentace, je výsledná forma výstupu z výpočtů genetického programování zdrojový kód, skládající se z příkazů.

V případě Stromové struktury ,jako na obrázku 1 , jsou pokyny uloženy formou matematických operací v uzlech daného stromu.Pozici listů zastávají vstupní proměnné. Výstupem po provedení výpočtu bude v tomto případě Strom reprezentující zobrazení matematické rovnice.

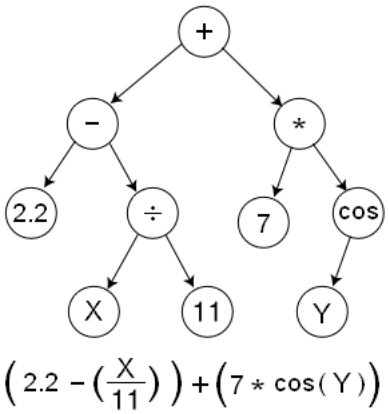
Pro využití v genetickém programování se převážně používá reprezentace formou stromové struktury protože se strom neskládá ze složitých operací ale jednoduchých matematických výrazů, což umožňuje snadnější ohodnocení programu a jeho evoluci.[2][3]

2.2.2 Algoritmus genetického programování

Jak již bylo zmíněno dříve, Genetické programování je určitá specializace Genetického algoritmu. V dnešní době již existuje mnoho variant Genetického algoritmu, a tak bude níže uvedena pouze jeho všeobecná popisná část ,vztahující se na Genetické programování.[2][3]

Verze tohoto Genetického algoritmu využívaného v genetickém programování se dá rozepsat do do několika kroků.

1. Inicializace - vytvoření populace.
2. Výběr jedinců z dané populace s vysokým Fitness ohodnocením.



Obrázek 1: Reprezentace programu pomocí stromové struktury.

3. Použitím evolučních metod, uvedených níže, vygeneruje z vybraných jedinců jedince nové Generace.
   - Mutace
   - Křížení
   - Reprodukce
4. Výpočet Fitness ohodnocení pro jedince nově vzniklé generace.
5. Otestování jedinců v případě nalezení ideálního jedince, je algoritmus ukončen, jinak pokračuje od bodu 2.
6. Ukončení a zobrazení výstupu algoritmu, v tomto případě jedinec s největším Fitness ohodnocením.

Inicializace - vytvoření populace
V prvním kroku genetického algoritmu určeného pro Genetické programování se většinou náhodně vygeneruje několik jedinců, kterým je přidělené náhodné ohodnocení. Této skupině jedinců se říká populace nulté generace. V případě genetického programování je každý jedinec reprezentován jednoduchým algoritmem obsahujícím matematické operace a vstupní proměnné. [2][3]

Výběr jedinců z populace
Z populace jedinců vzniklé v prvním kroku se vybere pár jedinců s velkým ohodnocením. Tento výběr většinou probíhá náhodně. Tito vybraní jedinci nadále postupují do další části, kdy jsou nad nimi zavolány určité evoluční techniky ,které provedou změnu daných jedinců.
[2][3]

Aplikace evolučních metod
V této části se nad vybranými jedinci aplikují evoluční techniky mutace, křížení popří-

padě reprodukce. Tyto techniky jsou podrobně rozepsány v seznamu uvedeném níže.

- **Křížení**
  V případě křížení se vezmou části dvou jedinců ,v případě Genetického programování jsou tito jedinci reprezentovaní Stromovou strukturou, takže taková to část může představovat část větve tohoto stromu. Spojením těchto části poté dojde k vytvoří nového jedince.
  [2][3]

- **Mutace**
  Mutace nevyžaduje dva jedince k jejímu provedení. Zmutovaný jedinec vzniká poté, co v jeho struktuře dojde ke změně nějakého genu. V případě Stromu tuto operaci představuje například záměna umístění uzlu ve stromu, popřípadě jeho rozmístění větví a listů.
  [2][3]

- **Reprodukce**
  Reprodukce neprovádí žádné změny nad daným jedincem. Jedinec se nezměněný přesune do další generace.
  [2][3]

  Otestování a ohodnocení Fitness
  Posledním krokem je otestování a přidělení ohodnocení nově vzniklým jedincům, kteří tvoří novou generaci. Toto ohodnoceni se přiřazuje na základě uživatelem specifikovaných kritérií, například schopnost vyřešit daný problém v určitém časovém intervalu. Pokud nějaký jedinec z nově vytvořené generace dosáhl kýženého ohodnocené, a je tedy schopen řešit danou úlohu dle stanovených kritérií, je prezentován na výstupu algoritmu a algoritmus se ukončuje. Pokud ideální jedinec nebyl nalezen v této generaci, proces dále pokračuje od bodu 2, dokud není nalezen ideální jedinec.
  [2][3]

3 Analýza systému ECJ

ECJ, tedy Java-based Evolutionary Computation Research System, je framework napsaný v jazyce Java poskytující širokou škálu reprezentace jedinců, chovných metod, fitness funkcí, evolučních algoritmů a paralelismu.

Systém ECJ obsahuje nepřeberné množství funkcí, které jsou rozděleny do patřičných knihoven, v případě programovacího jazyku Java balíčků. Strukturu tříd ECJ lze nalézt v příloze na konci tohoto dokumentu obrázek 9 a obrázek 10 .

Ovládání chování a nastavování funkce ECJ, je zařízeno pomocí konfiguračního souboru, který se předává jako jeden z parametrů pří spuštění. Tento soubor určuje chování celého ECJ, tzn. pokud budeme používat Genetické programování nebo Genetického algoritmu, bude to nastaveno v tomto konfiguračním souboru.

ECJ lze využít dvěma možnými způsoby, jako knihovnu ovládanou externím programem spouštěný kódem 2, který si jej volá dle potřeby nebo jako samostatný program spouštěný příkazem 1. V případě použití ECJ jako samostatného prvku se program spouští pomocí třídy *Evolve.class*, které musíme jako argument předat konfigurační soubor nastavující a ovládající chování ECJ. Při využití ECJ jako knihovny, kterou budeme vola z externí aplikace je potřeba vytvořit databázi parametrů, která je v předchozím případě nahrazená konfiguračním souborem. K vytvoření této databáze využijeme třídu *ParameterDatabase.class* obsaženou v části balíčku *ec.util* .[1]

```
java ec.Evolve -file soubor_s_parametry.params
```

Výpis 1: Příkaz na spuštění ECJ z konzole.

```
File parameterFile = ...
ParameterDatabase dbase = new ParameterDatabase(parameterFile,
new String[] { "-file", parameterFile.getCanonicalPath() });


EvolutionState evaluatedState = Evolve.initialize(child, 0, out);
evaluatedState.run(EvolutionState.C_STARTED_FRESH);
```

Výpis 2: Volání ECJ v programu

3.1 Výstup programu ECJ

Pří řešení problému v ECJ jak už pří užití Genetického programování nebo Genetického algoritmu systém generuje výstup daného procesu do souboru "out.stat"nacházejícího se ve složce ecj.ec a v případě potřeby také do konzole. V případě potřeby je možno program utišit a zakázat mu generování výstupu. Výstup vygenerovaný po provedení výpočtu obsahuje označení nově vytvořené generace, výpis ohodnocení Fitness nejlepších jedinců pravě vytvořené generace a počet Hits který značí kolik jedinců v dané generaci dosáhlo většího než průměrného ohodnocení. Na konci tohoto výpisu je vypsán nejlepší

jedinec tohoto výpočtu. Výstup do souboru *out.stat*, navíc obsahuje zápis algoritmů představujícího daného jedince textovou formou.[1]

Příkazy pro zakázaní generováni výstupu

- **stat.silent.print = true** - Zakáže výpis výstupu do konzole
- **stat.silent.file = true** - Zakáže výpis výstupu do souboru "out.stat"

3.2 Genetický algoritmus v ECJ

Systém ECJ se z velké částí soustředí a byl vytvořen pro Genetické programování, obsahuje ovšem i implementaci Genetického algoritmu a funkcí s ním spojených.

V této kapitole práce stručně popíše práci s Genetickým algoritmem v ECJ a způsob implementace tohoto algoritmu s ukázkovými částmi zdrojového kódu. Genetický algoritmus a jeho obecná funkce a fáze jeho vykonáváni ,byly již popsané v předchozí kapitole.

3.2.1 Reprezentace dat Genetického algoritmu v ECJ

V úvodní kapitole zabývající se obecným popisem Genetických algoritmů bylo již zmíněno že data popř. jedinci se kterými pracuje Genetický algoritmus, jsou nejčastěji tvoření skupinou bitů uspořádaných do jednorozměrného pole. ECJ ovšem poskytuje mnoho způsobů, jakými se dá daný jedinec zapsat. Taktéž poskytuje možnost pomocí přidělených metod vytvořit custom jedince obsahující prvky vícero datových typů např. *int*, *double*, *float*, *object*, *boolean*. V seznamu níže jsou uvedené reprezentace jedinců podporovaných ECJ. Každý z těchto jedinců je tvořen jedním vektorem uvedeného datového typu. [1]

Seznam použitelných datový typů pro reprezentaci jedince v GA ECJ

- **boolean** - jednorozměrné pole hodnot bool
  - Reprezentující třída: **ec.vector.BitVectorIndividual**
- **short** - jednorozměrné pole hodnot short
  - Reprezentující třída: **ec.vector.ShortVectorIndividual**
- **float** - jednorozměrné pole hodnot float
  - Reprezentující třída: **ec.vector.DoubleVectorIndividual**
- **double** - Jednorozměrné pole hodnot double.
  - Reprezentující třída: **ec.vector.DoubleVectorIndividual**

- **int** - Jednorozměrné pole hodnot int.
  - Reprezentující třída: **ec.vector.IntegerVectorIndividual**
- **byte** - Jednorozměrné pole hodnot byte.
  - Reprezentující třída: **ec.vector.ByteVectorIndividual**
- **long** - Jednorozměrné pole hodnot long.
  - Reprezentující třída: **ec.vector.LongVectorIndividual**
- **Podtřída ec.vector.Gene** - Slouží pro implementaci vlastního typu jedince.
  - Reprezentující třída: **ec.vector.GeneVectorIndividual**

3.2.2 Definováni problému pro Genetický algoritmus

Při výpočtu pomocí GA je potřeba definovat a implementovat třídu dědící ze třídy GP-Problem. Ve třídě GPProblem se nachází abstraktní metoda **evaluate** která bude obsahovat náš řešený problém.[1]

Příklad zdrojového kódu 3 implementace problému hledající takové uspořádání bitu, aby výsledné číslo reprezentované těmito bity dosahovalo co největší hodnoty.

```
public class MaxOnes extends Problem implements SimpleProblemForm
    {
    public void evaluate(final EvolutionState state,
        final Individual ind,
        final int subpopulation,
        final int threadnum)
        {
        if (ind.evaluated) return;

        if (!(ind instanceof BitVectorIndividual))
            state.output.fatal("It's not a BitVectorIndividual!!!",null);
        int sum=0;
        BitVectorIndividual ind2 = (BitVectorIndividual)ind;

        for(int x=0; x<ind2.genome.length; x++)
            sum += (ind2.genome[x] ? 1 : 0);

        if (!(ind2.fitness instanceof SimpleFitness))
            state.output.fatal("Whoa!  It's not a SimpleFitness!!!",null);
        ((SimpleFitness)ind2.fitness).setFitness(state,
            sum/(double)ind2.genome.length,
            sum == ind2.genome.length);
        ind2.evaluated = true;
        }
    }
```

Výpis 3: GA Třída implementující GPProblem MaxOnes

3.2.3 Výstup ECJ pro Genetický algoritmus

Pří, dokončení výpočtu ECJ vypíše podle nastavení výstup do souboru nebo konzole formou textu 4. Tento výstup obsahuje výpis všech generací, které byly vytvořeny v průběhu výpočtu a nejlepšího jedince z dané generace. V případě Genetického algoritmu je tento výstup ohodnocení Fitness daného jedince, hodnotu *bool* zda byl jedinec ohodnocen a sérii bitů představující jedince.[1]

```
Best Individual of Run:
Evaluated: true
Fitness: 1.0
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Výpis 4: Zobrazení výstupu GA ECJ do konzole

V případě výstupu uvedeném ve výpisu textu 4, se řešil problém MaxOnes tzn. nalézt takovou kombinací bitů, aby výsledné číslo reprezentované řetězcem těchto bitů mělo co největší možnou hodnotu. [1]

3.3 Genetické programování v ECJ

V této části se práce podrobně zaměří na popis a zdokumentování práce s Genetickým programováním v systému ECJ. V následující části bude podrobně rozebrána struktura a implementace Genetického programování v ECJ pomocí programovacího jazyku Java.

Procesy a postupy v systému ECJ, jsou z velké části převzaté popřípadě inspirované postupy nacházejícími se v *lil-gp Genetic Programming System*. Tento systém bude stručně popsán a srovnán se systémem ECJ v pozdější části této kapitoly. Velkou roli při tvorbě postupů nacházejících se v ECJ a způsobu využití biologických principů sehrál John R. Koza proto některé procesy, funkce a proměnné v ECJ nesou jeho jméno.[1]

3.3.1 Vytvoření třídy pro data výpočtu

Při tvorbě a průchodem stromu jsou postupně vykonávány operace specifikované v procházených uzlech. Tyto uzly obsahující matematické operace vyžadují pro svou funkcí vstupní data správného typu. Aby data předávána mezi jednotlivými uzly až do kořenového bodu odpovídala stejnému typu, je vytvořena speciální třída, ve které se specifikuje jaký typ dat se bude v daném stromu používat a předávat mezi jednotlivými prvky stromu při jeho průchodu. Tato uživatelem vytvořená třída musí dědit z třídy **ec.GPData** Příklad kódu této třídy je uveden níže 5. [1]

```
package ec.app.myapp;
import ec.gp.*;

public class MyData extends GPData
{
    public double val;
```

```
    public void copyTo(GPData other)
    {
        ((MyData)other).val = val; return other;
    }
}
```

Výpis 5: Příklad třídy pro uchování dat pro výpočet

3.3.2 Inicializace a popis tvorby jedinců

Jak již bylo zmíněno v předešlých kapitolách, v případě Genetického programování jsou jedinci reprezentováni stromovou strukturou ani v systému ECJ tomu není jinak. Příklad takového jedince je zobrazen na tomto obrázku 2 . Struktura tříd použitých pro tvorbu těchto stromů je zobrazená na obrázku 3 . Každý strom je tvořen kořenem, uzly a listy, které jsou v systému ECJ zastoupeny jednotlivými třídami.

Listy představují vstupní proměnné, které nemají žádného potomka jsou koncové prvky stromu. Uzly poté reprezentují různé matematické operace např. součet, sin, cos atd. jejíchž počet potomků se liší dle daného typu matematické operace např. u Uzlu funkce Sin bude potomek jen jeden naproti tomu u operace Součet můžou být potomci dva až tři, podle nastavení daného Uzlu. Počet potomků se nastavuje ve třídě *GPNode-Constraints* uvedené níže.Stavební prvky ze kterých se skládají stromy jsou v ECJ rozděleny na tři druhy. [1]

Rozdělení Uzlů/Listů v GP ECJ

- **Terminals** - jsou listy stromu, nemají potomky, jsou to vstupní proměnné.
- **Nonterminals** - jsou to Uzly obsahující matematické operace, mají jednoho až tři potomky.
- **Root** - Nejvyšší bod stromu, je to kořenový uzel , nemá žádné rodiče.

Struktura tříd reprezentujících strom a jejích rozložení.

- **GPInvididual** je třída reprezentující daného jednotlivce. Genetickém programování. Obsahuje:
  - **public GPTree[] trees** - pole stromů daného jedince, většinou obsahuje 1 strom.
- **GPTree** je třída zastupující datovou strukturu Strom. Obsahuje:
  - **public GPNode child** - odkaz na uzel potomka.
  - **public GPIndividual owner** - odkaz na jedince vlastnícího tento strom.
  - **public byte constraints** - odkaz na data vymezující vlastnosti daného Stromu.
- **GPNode** třída zastupující uzel nacházející se ve stromu. Obsahuje:

  - **public GPNodeParent parent** - hodnota ukazující na rodiče daného uzlu/-listu.
  - **public GPNode children[]** - seznam potomků tohoto uzlu.
  - **public byte argposition** - pozice tohoto prvku v rodičovském poli potomků.
  - **public byte constraints** - odkaz na data vymezující vlastnosti daného Stromu.

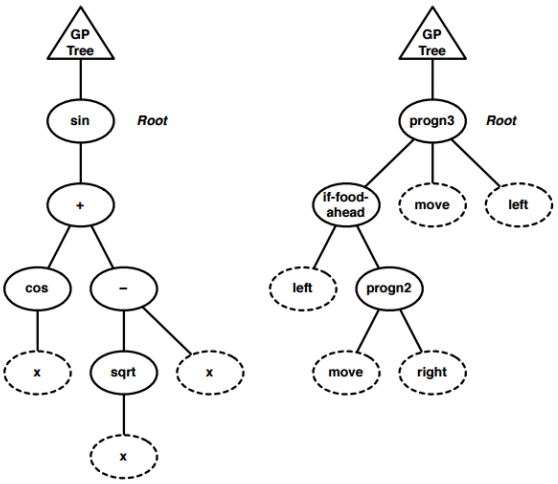
- **GPTreeConstraints** obsahuje data sdílena mezi stromy. Obsahuje:
  - **public byte constraintNumber** -počet stromů ukazující na tento objekt.
  - **public GPType treetype** - nastavuje typ stromu, návratová hodnota kořenu musí být stejného typu jako tento parametr.
  - **public String name** - obsahuje jméno instance této třídy.
  - **public GPNodeBuilder init** - obsahuje algoritmus zodpovědný za tvorbu tohoto stromu a jeho podstromu existuje více druhu algoritmu pro tvorbu stromu.
  - **public GPFunctionSet functionset** - obsahuje sadu funkcí pro tento strom např. klonování Uzlů.

- **GPNodeConstraints** obsahuje data sdílena mezi uzly. Obsahuje:
  - **public byte constraintNumber** - počet instancí této třídy
  - **public GPType returntype** - návratový typ Uzlu
  - **public GPType[] childtypes** -typy potomku Uzlu
  - **public String name** - obsahuje jméno instance této třídy
  - **public double probabilityOfSelection** -víceúčelová proměnná sloužící pro konstrukci stromu
  - **public GPNode zeroChildren[] = new GPNode[0]** -obsahuje instanci třídy GPNode bez potomku určenou pro Listy stromu

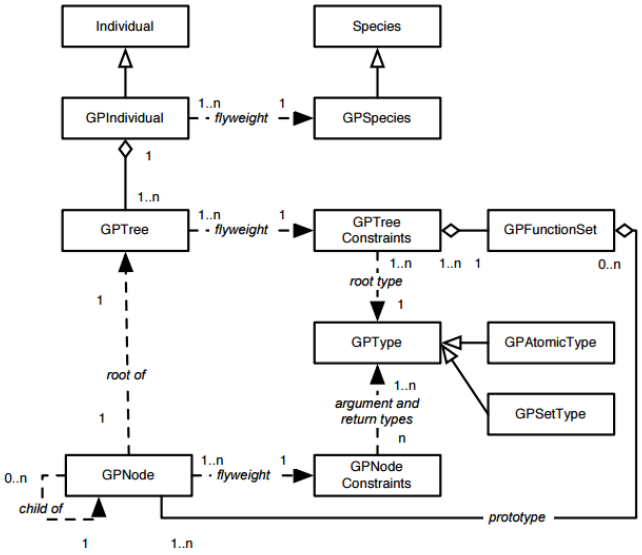
Před vytvořením jedince tzn. v Genetickém programování to je struktura strom je potřeba ručně implementovat, každý z prvků (Uzly/Listy), ze kterých se budou tvořit daní jedinci tzn stromová struktura algoritmu. Pro každý z těchto prvků tvořící strom ať už je to list nebo uzel, je vytvořená nová třída dědici z třídy GPNode. Uzly a listy tvořící tyto stromy mohou byt vstupní proměnné popř. matematické operace, jako sčítání,odečítání,násobeni atd[1].

Prvním typem funkcí jsou funkce s aritou 1. Tyto funkce obyčejně neprovádějí žádný výpočet jen předají dále hodnotu, kterou mají interně nastavenou. Příklad implementace kódu funkce arity 1 6.

```
package ec.app.myapp;
import ec.*;
import ec.gp.*;
```



Obrázek 2: Názorná ukázka stromu sestavených pomocí Genetického programování systému ECJ. [1]



Obrázek 3: Struktura tříd reprezentující jednotlivé části stromu způsob jakým je strom sestaven. [1]

```
public class X extends GPNode {
    // string pouzity pro tisk pri graficke zobrazeni stromu
  public String toString () { return "x"; };
  //pocet potomku
  public int expectedChildren() { return 0; }
```

```
    //funkce pro vypocet – preda jen prednasatvene data
    public void eval(EvolutionState state,
    int thread,
    GPData input,
    ADFStack stack,
    GPIndividual indivdiual,
    GPProblem problem) {

    MyData data = (MyData) input;

    MyProblem prob = (MyProblem) problem;

    data.val = problem.Xs[problem.current];
}
}
```

Výpis 6: Implementace třídy arity 1.

Dalším typem funkcí jsou funkce arity 2 obvykle vykonávající nějakou operaci vyžadující vnější data. Příklad takové funkce sčítající dvě proměnné je uveden zde 7.

```
package ec.app.myapp;
import ec.*;
import ec.gp.*;

public class Add extends GPNode {
    // prirazeni oznaceni operace pro pozdejsi graficke zobrazeni
    public String toString() { return "+"; };
    //operace souctu ma 2 vstupni promenne
    public int expectedChildren() { return 2; }

    public void eval(EvolutionState state,
                int thread, GPData input,
                ADFStack stack,
                GPIndividual individual,
                GPProblem problem) {

        MyData data = (MyData) input;
        MyProblem prob = (MyProblem) problem;
        children[0].eval(state, thread, data,          stack, individual, prob);
        double val1 = data.val;
        children[1].eval(state, thread, data,  stack, individual, prob);
        //secteni dvou vstupnich promennych
        data.val = val1 + data.val;
}
}
```

Výpis 7: Příklad implementace matematické operace - Uzlu stromu.

Velkou nevýhodou ECJ je jeho věk, neposkytuje totiž žádné uživatelsky přívětivé uživatelské rozhraní, které by umožňovalo zadávaní prvků pro tvorbu Stromu bez nutnosti zdlouhavého psaní kódu. V případě, kdy do problému, který chceme řešit vstupuje mnoho proměnných a obsahuje veliké množství použitelných matematických operací, se tak zvyšuje časová náročnost pro uživatele.

3.3.3 Konstrukce stromů

V předešlé části této kapitoly bylo vysvětleno, jak se tvoří třídy představující Uzly/Listy pro tvorbu stromu. Tato část kapitoly se zaměří na konstrukcí těchto Stromů, které reprezentují jedince v GP. ECJ obsahuje mnoho metod pro tvorbu těchto stromů, každá lišící se způsobem tvorby těchto stromů a možnostmi nastavení parametrů pro tvořený strom např. maximální/minimální hloubka stromu a velikost stromu v počtu uzlů a listů popř. přiřazení pravděpodobnosti, s jakou bude daný list/uzel použit při tvorbě stromu. [1]

Algoritmy pro konstrukci stromů

- **ec.gp.koza.FullBuilder** Tvoří celé stromy pomoci Kozového FULL algoritmu. Nelze u něj zvolit velikost stromu. Vstupní parametry jsou minimální a maximální hloubka stromu.
- **ec.gp.koza.HalfBuilder** HalfBuilder je implicitní algoritmus pro tvorbu stromu v ECJ. Pomocí hodu mince rozhodne pokud pro tvorbu využije GROW nebo FULL builder. Vstupní parametry jsou minimální a maximální hloubka stromu a šance na vykonání funkce GROW.Příklad nastavení *HalfBuilder* metody v konfiguračním souboru 8.
- **ec.gp.koza.GrowBuilder** Vytvoří libovolný strom pomocí Kozového GROW algoritmu. Nelze u něj zvolit maximální velikost. Vstupní parametry jsou minimální a maximální hloubka stromu.
- **ec.gp.build.PTC1** PTC1 je modifikovaná verze GROW algoritmu která zaručuje že strom bude vytvořen podle daného kontextu. Nejde u něj zvolit velikost stromu který vytvoří ale umožňuje stanovit pravděpodobnost vybrání daného uzlu nebo listu při tvorbě stromu. Vstupní parametry tohoto algoritmu jsou očekávaná velikost stromu a maximální hloubka.

```
gp.tc.0. init  = ec.gp.koza.HalfBuilder
gp.tc.0. init .min-depth = 2
gp.tc.0. init .max-depth = 6
gp.tc.0. init .growp = 0.5
```

Výpis 8: Nastavení algoritmu pro tvorbu stromu HalfBuilder.

3.3.4 Definování,implementace problému a Fitness funkce

Problém nebo úlohu, kterou chceme pomocí Genetického programování řešit, je potřeba definovat a naprogramovat. Složitost námi řešeného problému zásadně ovlivňuje rychlost, s jakou bude ECJ schopen provést kompletní výpočet. Pří psaní kódu, který bude simulovat náš problém a testovat výkonnost daného jedince musíme brát v potaz, že daný kód bude opakovaně vykonáván nad velkým množstvím algoritmů. Kód by proto měl mít co nejmenší složitost. Třída obsahující náš problém musí dědit z třídy *ec.gp.GPProblem*

obsahující abstraktní metodu **evaluate**, která je zodpovědná za výpočty při testování jednotlivých jedinců a jejich ohodnocování. V příkladu uvedeném níže 9 ,je zobrazena metoda řešící problém Pythagorovy věty. Cílem této metody je nalézt rovnici takovou, aby výsledek této rovnice po dosazení hodnot odpovídal výsledku Pythagorovy rovnice. Nalezení této rovnice dosáhneme tak, že vytvoříme určité množství rovnic a výsledek vypočítaný těmito rovnicemi porovnáme s výsledkem správné Pythagorovy věty. Pokud se rozdíl těchto výsledku rovná nule výpočet je ukončen a byla úspěšně nalezena námi požadovaná rovnice. [1]

```
package ec.app.PythagorasProblem;
import ec.util.*;
import ec.*;
import ec.gp.*;
import ec.gp.koza.*;
import ec.simple.*;

public class PythagorasProblem extends GPProblem implements SimpleProblemForm
    {
    private static final long serialVersionUID = 1;

    public double X;
    public double Y;

    public void setup(final EvolutionState state,
        final Parameter base)
        {
        super.setup(state, base);
        if (!( input instanceof DoubleData))
            state.output.fatal ("GPData␣class␣must␣subclass␣from␣" + DoubleData.class,
                base.push(P_DATA), null);
        }

    public void evaluate(final EvolutionState state,
        final Individual ind,
        final int subpopulation,
        final int threadnum)
        {
        if (!ind.evaluated)
            {
            DoubleData input = (DoubleData)(this.input);

            int hits = 0;
            double sum = 0.0;
            double expectedResult;
            double result;

                // prirazeni nahodnych hodnot do promenych
                X = state.random[threadnum].nextDouble() * 10;
                Y = state.random[threadnum].nextDouble() * 10;

                //provedeni vypoctu Pythagorovy vety
```

```
            expectedResult = X*X + Y*Y;
        //zavolani funkce ktera provede vypocet pomogi vygenerovaneho algoritmu
            ((GPIndividual)ind).trees[0].child.eval(
                state,threadnum,input,stack,((GPIndividual)ind),this);

        //ziskani vysledku algoritmu
            result = input.x;
            if (result <= 0.01) hits++;

            // rozdil vysledku zadane Pythagorovy rovnice a rovnice vygenerovane pomoci GA
                - pokud se rozdil = 0 byla nalezena spravna rovnice
            Math.abs(expectedResult - result) / 100;
            sum = ;

        KozaFitness f = ((KozaFitness)ind.fitness);
        f.setStandardizedFitness(state, sum);
        f.hits = hits;
        ind.evaluated = true;
        }
    }
}
```

Výpis 9: Implementace třídy PythagorasProblem

3.3.5 Šlechtící techniky ECJ v GP

ECJ obsahuje mnoho šlechtících technik. Nejpoužívanější techniky jsou, Kozovo křížení *ec.gp.koza.Crossover* a mutace *ec.gp.koza.Mutation*. Další techniky lze najít v balíčku *ec.gp.breed*. Metody zajišťující šlechtění vybírají převážně jediný strom daného jedince nad kterým se vykonají operace mutace nebo křížení. Ve většině případů lze nastavit nad kterým stromem se tyto operace provedou nebo nechat metody šlechtění aby tyto stromy vybíraly náhodně. Jakmile je vybrán strom nad kterým se provede šlechtici operace musí se vybrat nad kterým prvkem stromu budou tyto operace křížení nebo mutace provedeny. K tomuto účelu slouží interface ec.gp.GPNodeSelector obsahující metodu pro výběr uzlu-/listu. Standardním *GPNodeSelectorem* je *ec.gp.koza.KozaNodeSelector* který vybírá uzly/-listy podle toho jakou mají přiřazenou pravděpodobnost výběru.[1]

Ukázka nastavení pravděpodobností výběru daných částí stromu10. *Terminals* - listy stromu obsahující vstupní proměnné, *Nonterminals* - Uzly obsahující matematické operace např. součet, *Root* - kořenový uzel.

```
gp.koza.ns.terminals = 0.1
gp.koza.ns.nonterminals = 0.9
gp.koza.ns.root = 0.0
```

Výpis 10: Nastavení parametru křížení v GP ECJ.

Tento zápis 10 říká, že listy stromu mají 10 % šanci že budou vybrány pro křížení nebo mutaci, uzly obsahující matematické operace mají 90 % šanci na výběr a kořen stromu má

cíleně stanovenou tuto šanci na 0 %.

Funkce křížení v ECJ GP - ec.gp.koza.CrossoverPipeline
Funkce křížení v GP provádí klasické křížení dvou podstromů. Ze dvou zdrojových jedinců vezme jejich stromy, ve kterých vybere uzly a poté přehodí podstromy, které mají kořen ve vybraných uzlech. Tímto vzniknou dva nové stromy.Vlastnosti křížení se dají změnit pomocí příkazu 11 v konfiguračním souboru.[1]

```
pop.subpop.0.species.pipe.tries = 1
pop.subpop.0.species.pipe.maxdepth = 17
pop.subpop.0.species.pipe.toss = false
pop.subpop.0.species.pipe.maxsize = ...
gp.koza.xover.tree.0
gp.koza.xover.tree.1
pop.subpop.0.species.pipe.ns = ec.gp.koza.KozaNodeSelector
```

Výpis 11: Nastavení parametru křížení v GP ECJ.

Pomocí prvního příkazu se nastavuje pokus vyhledání uzlů splňujícího dané podmínky. Pokud není nalezen, funkce vrátí nezměněné rodiče. Druhý parametr specifikuje, že potomek vzniklý křížením nesmí překročit hloubku 17 uzlů. Třetí příkaz říká v případě hodnoty false že budou vráceni oba potomci. V případě nastavení této hodnoty na *true* bude navracen pouze jediný potomek a druhý bude zahozen. Čtvrtý parametr je volitelný a není implicitně definovaný. Tento parametr slouží k nastavení maximální velikosti potomků. V neposlední řadě toto nastavení obsahuje příkazy pro výběr stromů nad kterými se provede křížení a volbu metody která provede výběr uzlu z rodičovských stromů. [1]

Funkce Mutace v ECJ GP - ec.gp.koza.MutationPipeline
Mutace v GP probíhá tak, že je vybrán jedinec z tohoto jedince jeho strom. V tomto stromu se vybere uzel a podstrom, který má kořen v tomto vybraném uzlu je nahrazen kompletně novým náhodně vygenerovaným podstromem. Možnosti mutace lze měnit pomocí příkazu 12 v konfiguračním souboru.[1]

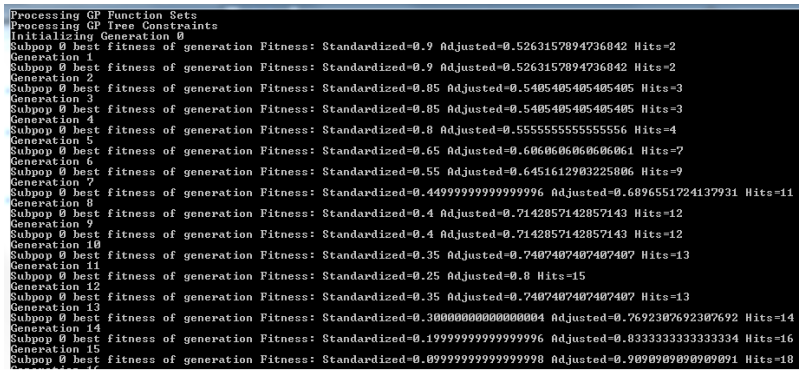
```
pop.subpop.0.species.pipe.tries = 1
pop.subpop.0.species.pipe.maxdepth = 17
pop.subpop.0.species.pipe.tree.0 = 0
pop.subpop.0.species.pipe.ns = ec.gp.koza.KozaNodeSelector
pop.subpop.0.species.pipe.maxsize = ...
gp.koza.mutate.build.0 = ec.gp.koza.GrowBuilder
gp.koza.mutate.equal = true
```

Výpis 12: Nastavení parametrů Mutace v GP ECJ.

Parametr **gp.koza.mutate.build.0 = ec.gp.koza.GrowBuilder** určuje, jaká metoda bude použita pro tvorbu náhodně generovaných podstromů. Parametr **gp.koza.mutate.equal = true** slouží ke stanovení podmínky, že nově vytvořený podstrom musí být stejné velikosti jako starý nahrazený podstrom.

3.3.6 Výstup Genetického programování ECJ

Výstup Genetického programování ECJ se značně liší oproti genetickému algoritmu, jelikož cílem GP je získat určitý algoritmus. Níže je uveden příklad algoritmu vygenerovaného pomocí GP 15. Tento algoritmus obsahuje funkce a vstupní proměnné na rozdíl od Genetického algoritmu, kdy výstupem byla převážně série bitů. Tento textový výstup je generován do souboru **out.stat**, ve kterém se nachází všechny generace a nejlepší jedinci z dané generace.Dále je zde uveden příklad výstupu do konzole 4. Výstup do konzole na rozdíl od výstupu do souboru nezobrazuje výsledný algoritmus. [1]



Obrázek 4: Výstup do konzole - ECJ GP

```
(ifDangerAhead LEFT (ifFoodRight (ifMovingRight
    (ifMovingLeft  RIGHT (ifDangerAhead LEFT LEFT))
    (ifFoodUp (ifDangerAhead LEFT FORWARD) (ifMovingLeft
        RIGHT FORWARD))) (ifFoodUp FORWARD (ifFoodRight
    (ifFoodRight FORWARD RIGHT) (ifMovingLeft
    FORWARD FORWARD)))))
```

Výpis 13: Příklad jednoduchého algoritmu vygenerovaného pomocí Genetického programování ECJ..

3.3.7 Paralelizmus

V mnoha případech ECJ podporuje několik vláken ve dvou fázích evolučního procesu. Tyto procesy jsou šlechtění a ohodnocování. Pomocí konfiguračního souboru můžeme specifikovat 14 kolik vláken chceme použít pro některou z těchto dvou fází evolučního procesu. [1]

```
breedthreads = 4
evalthreads = 4
```

Výpis 14: Nastavení počtu vláken pro výpočet.

Tyto příkazy specifikovaly, že pro každou fází, evolučního procesu, kde je možné využít více vláken byly vytvořeny právě čtyři vlákna. Počet těchto vláken se zpravidla odvozuje od počtu jader procesoru.[1]

3.3.8 Distribuované ohodnocení

Distribuované ohodnocení funguje na základě jednoho Master ECJ procesu a N Slave ECJ procesu. Master ovládá evoluční smyčku, a když je potřeba ohodnotit jedince dané populace rozešle je na Slave ECJ procesy k vyhodnocení. Tímto způsobem může probíhat ohodnocování jedinců paralelně. Distribuované ohodnocení je výhodné, jedině tehdy, pokud čas ušetřený paralelizaci předčí čas ztrávený rozesíláním jedinců přes síť do Slave jednotek, ohodnocováním a zpětným odesláním jedinců.[1]

3.3.9 Island models

Island model umožňuje řešit stejný problém na několika nezávislých strojích propojených pomocí sítě. V případě nalezení nějakého speciálně výkonného jedince, je možné jej přeposlat na jinou stanici, kde bude využit pro další výpočet. [1]

3.3.10 Alternativní řešení pro evoluční výpočty

V této kapitole se nachází seznam nástrojů pro evoluční výpočty, jejich stručný popis a výhody které nabízejí oproti systému ECJ. V mnoha ohledech dostupné platformy poskytují stejné funkce jako ECJ a liší se pouze způsobem implementace.

ECF - Evolutionary Computation Framework
ECJ je framework pro evoluční výpočty napsaný v programovacím jazyce C++. Tento framework obsahuje všechny funkce, které lze nalézt v systému ECJ včetně paralelizmu. Výhodou tohoto frameworku je možnost konfigurace algoritmů, genů a parametrů bez nutnosti rekompilace.

GPE - Genetic Programming Engine
GPE je program umožňující využití techniky genetického programování, určený pro platformu .NET. Tento systém, jako jediný z uvedených, obsahuje plně funkční uživatelský interface umožňující v průběhu výpočtu zobrazovat aktuálně zpracovávaná data např. chování konkrétního algoritmu a statistiku.

Watchmaker Framework for Evolutionary Computation
WF framework je další platforma pro evoluční výpočty, napsaná v programovacím jazyku JAVA. Framework obsahuje totožné funkce jako ECJ. Výhodou WF je neinvazivní implementace tříd, určených pro výpočet. Třídy nemusí dědit z předdefinovaných tříd ani nemusí implementovat rozhraní, díky tomu nejsou při tvorbě programu kladena žádná omezení.

4 Popis a dokumentace aplikace

V následné kapitole bude podrobně zdokumentovaná aplikace využívající Genetické programování ECJ pro výpočet vzorce určující chování umělé inteligence. Tato aplikace, stejně jako systém ECJ, je implementovaná v jazyce Java.

4.1 Stručný popis a funkce aplikace

Pro využití a aplikaci Genetického programování byla vybrána hra Snake. Inspirace k řešení hry Snake byla převzata z článku [5] . V této hře se pohybuje had po herní ploše a sbírá kusy jídla. Podle počtu sesbíraných kusů jídla se poté určí skóre. Jedním z problému tohoto postupu tvorby algoritmu pro ovládání hada je, že ne vždy se podaří nalézt vhodného jedince po první spuštění programu. Mnohdy nastává situace, kdy ideální jedinec byl nalezen až po několika opakovaných spuštěních programu. Důvod implementace ECJ tímto způsobem, kdy program opakovaně pří každém spuštění provádí výpočet pro nalezení algoritmu na ovládáni hada je neschopnost ECJ jednoduchým způsobem zrekonstruovat strukturu tříd ze souboru, která reprezentuje ovládací algoritmus UI ve hře. Tento problém vznikl z důvodu velké složitosti ECJ, množství tříd a jejích provázanosti viz Struktura tříd ECJ 9 10. V kapitole Přílohy se nachází sekvenční diagram 11 , zobrazující postupný průběh výpočtu ECJ, při tvorbě algoritmu ovládajících hada a jejich testování.

Jak rychle se program vykoná a nalezne jedince záleží, jak velkou populaci a kolik generací musí vytvořit a otestovat. S rostoucím počtem generací a jedinců v dané generaci se taktéž zvyšuje čas dokončení výpočtu. V případě slabšího hardware, může dokončení tohoto výpočtu trvat i několik minut. Pokud je potřeba tento čas snížit ,je možno zredukovat množství jedinců v generaci a počet generací na úkor toho, že se sníží šance na nalezení ideálního jedince, což vede k opětovnému spuštění programu. Další z možností jak zlepšit časovou náročnost je možnost omezit počet funkcí pro konstrukci algoritmu nebo velikost a hloubku konstruovaných stromů. Tímto ovšem může nastat situace že nebude možné sestrojit jedince schopného vyřešit náš problém se zadanými parametry tzn. posbírat stanovený počet kusu jídla.

Tato úloha je vhodná pro řešení pomocí genetického programování, kdy pomocí GP sestrojíme Strom (Algoritmus), který nám daného hada ovládá a naviguje ho postupně po herní ploše. Čím více kousků jídla had ovládaný daným algoritmem nasbírá, tím větší má algoritmus ovládající tohoto hada výkonnost. Tato výkonnost se označuje Fitness. Implementace této hry měla obsahovat taktéž náhodně generované překážky. Vzhledem k tomu že se při přidání překážek zvýšila mnohonásobně časová náročnost na dokončení výpočtu nebyly nakonec obsaženy ve výsledné hře.

Samostatný had se pohybuje po herní ploše která je tvořena pomocí souřadnicové mřížky toto umožňuje přesně sledovat pohyb hada a předávat přesné souřadnice do funkcí starajících se o vyhodnocování kterým směrem se had v dalším kroku vydá. V případě, že had narazí do překážky nebo posbírá všechny kousky jídla je hra ukončena.

4.2 Inicializace a spuštění programu

Ke správnému chodu je potřeba mít v počítači nainstalovanou správnou verzi JAVA Runtime Enviorment 1.8.0. Před každým spuštěním této aplikace se spustí ECJ, které podle zadaných parametrů provede výpočet a najde nejvhodnější algoritmus pro ovládání UI hada ve hře. Výsledný algoritmus se pří každém spuštění ECJ liší, UI ovládající hada proto není pokaždé stejná a jeho chování tak bývá nepředvídatelné. Po nalezení algoritmu se zobrazí hlavní menu.

Spuštění programů probíhá přes příkazový řádek. Před spuštěním je potřeba nastavit CLASSPATH 15. Do hodnoty CLASSPATH se nastavuje adresa složky ne které se nachází soubory *SnakeRun.class*, *BP_1_params.params* a složka *ec*. Po nastavení proměnné PATH se had spustí pomocí příkazu 16.

```
set CLASSPATH=C:\Plocha\BP_KOU0060
```

Výpis 15: Příklad příkazu pro nastavení CLASSPATH.

```
java SnakeRun
```

Výpis 16: Spuštění hry Snake.

V případě že není v operačním systém nastavená proměnná PATH určující cestu ke složce obsahující JAVA Runtime Enviorment musí uživatel taktéž zadat cestu k tomuto souboru 17.

```
C:\Java\jdk1.7.0\bin\java SnakeRun
```

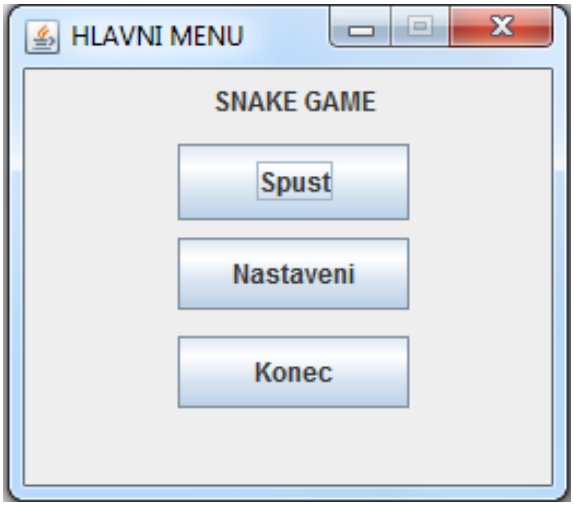
Výpis 17: Spuštění hry Snake bez nastavení cesty PATH.

4.3 Popis UI a ovládacích prvků

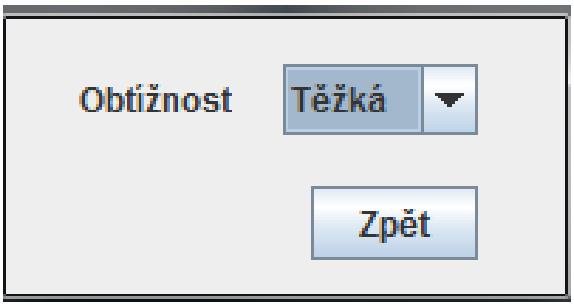
Program obsahuje velmi jednoduché uživatelské rozhraní. Ke, zobrazení uživatelského rozhraní dojde až poté, kdy ECJ dokončí výpočet a vytvoří algoritmus řídící počítačem ovládaného hada.

Hlavní menu
Po dokončení hledání algoritmu pro ovládání hada, se zobrazí okno s hlavním menu 5. Toto okno obsahuje tlačítko pro zahájení hry, nastavení a ukončení programu.

Nastaveni
Okno Nastaveni 6 si uživatel zobrazí pomocí tlačítka nastaveni v *Hlavním menu*. Toto okno slouží pro nastavení obtížnosti hry. Obtížnost určuje jak rychle se bude had pohybovat po herní ploše. Hra je v základu nastavená na nejtěžší obtížnost.



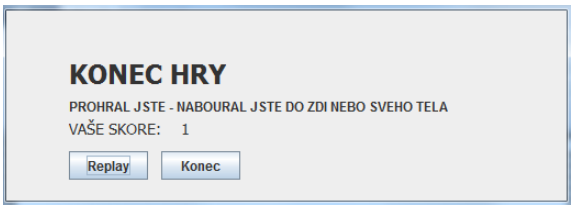
Obrázek 5: Hlavní menu



Obrázek 6: Nastaveni

Obrazovka pro ukončení hry

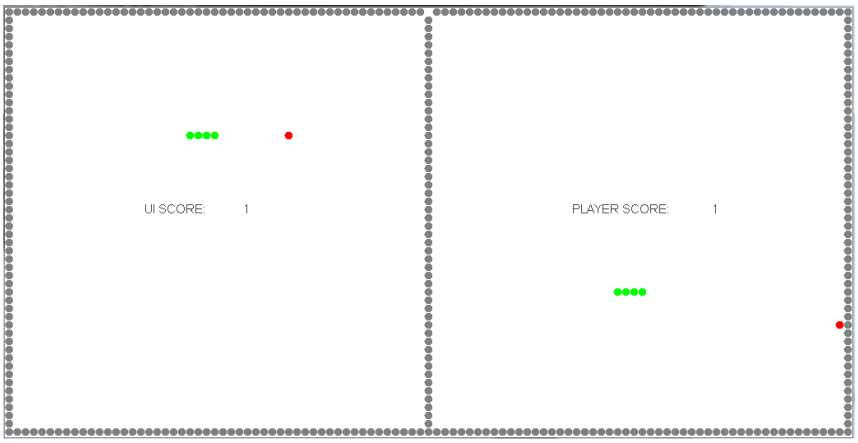
V případě, že hráč nebo počítačem ovládaný had posbírá všechny kousky jídla nebo narazí do překážky, je hra okamžitě ukončena a zobrazena obrazovka obsahující informace o příčině ukončení hry a dosažené skore 7. Toto okno obsahuje dvě tlačítka. Tlačítko *Konec ukončí* program a tlačítko Replay znovu zahájí hru proti počítači.



Obrázek 7: Okno zobrazené po ukončení hry

Herní plocha

Herní plocha 8 je rozdělena na dvě poloviny. Ve středu každé poloviny herní plochy je zobrazeno *score* patřičného hada.V levé polovině herní plochy se pohybuje had ovládaný umělou inteligencí vygenerovanou pomocí ECJ a na pravé straně herní plochy se nachází hráčem ovládaný had. Herní pole po kterém se had může pohybovat je ohraničeno zdí tvořenou šedými kruhy. Pokud hráč nebo had ovládaný počítačem do této zdi narazí, hra je okamžitě ukončena. Červené kruhy představují jídlo které had sbírá.



Obrázek 8: Herní plocha

4.4 Implementace tříd pro tvorbu stromu algoritmu

Jak již bylo řečeno pří řešení problému v ECJ genetickém programování, je potřeba vytvořit stavební bloky (Uzly a Listy,) ze kterých se budou vytvářet daní jedinci (Strom), který vytvoří výsledný algoritmus. Tento algoritmus se skládá ze série vstupních proměnných a podmínek.

První důležitou třídou je třída **Direction** 18, sloužící pro předávání dat mezi jednotlivými uzly. Tato třída obsahuje informace o aktuální pozici hlavy ,hada a jídla. Tyto data jsou poté předávané touto třídou z potomků do rodičů ,spolu s výsledným směrem který určila daná funkce.

```
package ec.app.myapp;
import ec.util.*;
import ec.*;
import ec.gp.*;
import java.util.ArrayList;

public class Direction extends GPData
    {
    //smery kterymi se had muze vydat
    public enum Directions {LEFT,RIGHT,FORWARD,UP,DOWN};
```

```
//souradnice kousku jidla
public int foodX;
public int foodY;
//souradnice hlavy hada
public int snakeHeadX;
public int snakeHeadY;
//smer ktery byl vypocitan funkci ktera tuto instanci predava svemu predku
public Directions smer;

public void copyTo(final GPData gpd) // copy my stuff to another DoubleData
    { ((Direction)gpd).smer = smer; }
}
```

Výpis 18: Datová třída Direction

Při tvorbě stromu jsou využity dva typy funkcí funkce s aritou 1 a funkce s aritou 2. Funkce s aritou 1 neprovádějí žádný výpočet, jenom předají směr ,který jim byl určen. Příklad implementace třídy obsahující funkci arity 1 19. Tyto funkce zastávají roli listu stromu. Funkce s aritou 2 mají dva stupy tzn. dva různé směry, které může funkce po provedení výpočtu vrátit.

```
package ec.app.myapp;
import ec.*;
import ec.gp.*;
import ec.util.*;

public class Right extends GPNode
    {
    public String toString() { return "RIGHT"; }

    public int expectedChildren() { return 0; }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem)
        {
        Direction rd = ((Direction)(input));
        rd.smer = ((SnakeProblem)problem).had.smerChange(Direction.Directions.RIGHT);;
        rd.foodX = ((SnakeProblem)problem).foodX;
        rd.foodY = ((SnakeProblem)problem).foodY;
        rd.snakeHeadX = ((SnakeProblem)problem).snakeHeadX;
        rd.snakeHeadY = ((SnakeProblem)problem).snakeHeadY;
        }
    }
```

Výpis 19: Funkce arity 1 RIGHT

Seznam funkcí arity 1

- **FORWARD** Vstupní směr FUNKCE - pokračuj v aktuální směru

- **LEFT** Vstupní směr FUNKCE - V dalším kroku zatoč vlevo od aktuálního směru
- **RIGHT** Vstupní směr FUNKCE - V dalším kroku zatoč vpravo od aktuálního směru

Funkce pro ovládání směru s aritou 2 - Uzly
Implementace stavebních prvků, pro tvorbu algoritmu, pro ovládaní hada obsahuje mnoho funkcí, které vyhodnocují dané podmínky a podle toho rozhodují ,jestli se had vydá určitým směrem. Každá tato funkce má dva vstupy, tzn. dva různé směry, kterými se had po vyhodnocení funkce může vydat. Funkce po vyhodnocení přepošle do svého rodiče jeden ze směr,ů kterými jsou potomky dané funkce. Samotné vyhodnocování funkce probíhá pomocí dat předaných ve třídě *Direction*, které obsahuje aktuální informace o poloze hada, sloužící pro vyhodnocení podmínky. Příklad takovéto funkce je uveden níže 20, v tomto případě je to funkce pro výpočet zdali se ve směru hada nachází nějaké nebezpečí např. zeď nebo jeho část těla. Každá funkce, uvedená v seznamu musí byt implementovaná v samostatné třídě.

```
package ec.app.myapp;
import ec.*;
import ec.gp.*;
import ec.util.*;

public class ifDangerAhead extends GPNode
    {
    public String toString() { return "ifDangerAhead"; }

    public int expectedChildren() { return 2; }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem)
        {

        Direction rd = ((Direction)(input));
        Direction.Directions smer;

        children[0].eval(state,thread,input,stack,individual,problem);
        smer = rd.smer;
        children[1].eval(state,thread,input,stack,individual,problem);

        Cords souradnice = SnakeProblem.getDirection(((SnakeProblem)problem).had.smer,rd.
            snakeHeadX,rd.snakeHeadY);

        if (souradnice.posX > 50 || souradnice.posX < 1 || souradnice.posY > 50 || souradnice.posY
            < 1)
        {
            rd.smer = smer;
        }
```

```
        for(int i = ((SnakeProblem)problem).had.telo.size() − 1; i > 0; i−−)
        {
            if (souradnice.posX == ((SnakeProblem)problem).had.telo.get(i).posX && souradnice.
                posY == ((SnakeProblem)problem).had.telo.get(i).posY)
            {
                rd.smer = smer;
            }
        }
        }
    }
```

Výpis 20: Funkce arity 2 ifDangerAhead zjišťující zdali se jídlo nachází ve směru pohybu hada.

Kompletní výpis funkcí ovládající směr hada

- **ifMoovingUp** - Pokud se had pohybuje nahoru, změní had směr předaný POTOMKEM1, jinak změní směr předaný POTOMKEM2.
- **ifMoovingDown** - Pokud se had pohybuje směrem dolů, změní had směr předaný POTOMKEM1, jinak změní směr předaný POTOMKEM2.
- **ifMoovingLeft** - Pokud se had pohybuje směrem vlevo, změní had směr předaný POTOMKEM1, jinak změní směr předaný POTOMKEM2.
- **ifMoovingRight** - Pokud se had pohybuje směrem vpravo, změní had směr předaný POTOMKEM1, jinak změní směr předaný POTOMKEM2.
- **ifFoodAhead** - Pokud se ve směru pohybu hada nachází jídlo, had změní směr předaný POTOMKEM1, jinak změní směr předaný POTOMKEM2.
- **ifDangerLeft** - Pokud se po levé straně hadovy hlavy nachází zeď, nebo část jeho těla, had změní směr předaný POTOMKEM1, jinak změní směr předaný POTOMKEM2.
- **ifDangerRight** - Pokud se po pravé straně hadovy hlavy nachází zeď nebo část jeho těla, had změní směr předaný POTOMKEM1, jinak změní směr předaný POTOMKEM2.
- **ifDangerAhead** - Pokud se před hlavou hada nachází zeď nebo část jeho těla, had změní směr předaný POTOMKEM1, jinak změní směr předaný POTOMKEM2.
- **ifDangerTwoAhead** - Pokud se dvě pole před hlavou hada nachází zeď nebo část jeho těla, had změní směr předaný POTOMKEM1, jinak změní směr předaný POTOMKEM2.
- **ifFoodRight** - Pokud se jídlo nachází blíže pravé strany herní ploch než hlava hada, had změní směr předaný POTOMKEM1, jinak změní směr předaný POTOMKEM2.
- **ifFoodUp** - Pokud se jídlo nachází blíže horní strany herní ploch než hlava hada, had změní směr předaný POTOMKEM1, jinak změní směr předaný POTOMKEM2.

4.5 Implementace problému SnakeProblem

Tato část se zabývá implementaci řečeného problému, v tomto případě se snažíme nalézt algoritmus který je schopen navigovat hada po herní ploše tak, aby sesbíral co nejvíce kusu jídla. V předchozí části byl vysvětlen vznik těchto algoritmů a co obsahují. Tato část se zabývá implementací třídy obsahující metody pro testování vzniklých algoritmů a nalezení toho nejlepšího 21.

Testování probíhá pomocí simulace hry, kdy pro každý algoritmus je vytvořen nový had a umístěn do herního pole, poté se zavolá daný algoritmus, který počítá v každé iteraci dané smyčky jakým směrem se má had pohnout. Pokud had pří pohybu narazí na kus jídla jeho velikost se zvětší zkonzumovaný kus jídla je smazán a na herní plochu je přidán kus nový. V případě že had posbíral určitý počet kousku jídla nebo narazil do zdi popř. do části svého těla je smyčka ukončena a danému algoritmu přiřazeno ohodnocení podle toho kolik kousku zkonzumoval. Simulace je v základu nastavená na test pro 20 kousků jídla. Pokud se algoritmu v dané simulaci podaří sesbírat všech 20 kousků jídla je tento algoritmus ohodnocen Fitness 0.0 v případě že se mu nepodaří sebrat jediný kus jídla je ohodnocení 1.0.

```
public void evaluate(final EvolutionState state,
                     final Individual ind,
                     final int subpopulation,
                     final int threadnum)
{
if (true)
        {

Direction input = (Direction)(this.input);

ArrayList<Food> jidlo = new ArrayList<Food>();
int score = 0;
had = new Snake();
Food food = makeFood();

jidlo.add(food);

for(int m = 0; m < 1000;m++)
{
    snakeHeadX = had.telo.get(0).posX;
    snakeHeadY = had.telo.get(0).posY;

    if(jidlo.isEmpty()) //pokud neni jidlo , vygeneruj nove
    {
        food = makeFood();
        jidlo.add(food);
    }
    foodX = jidlo.get(0).posX;
    foodY = jidlo.get(0).posY;

   //zavolani algoritmu pro vypocet smeru hada
```

```
((GPIndividual)ind).trees[0].child.eval(state,threadnum,input,stack,((GPIndividual)ind),
    this);

  //nastaveni noveho smeru hada
  had.smer = input.smer;

  //pokud jsou vsechny kousky jidla sesbirane ukonci simulaci
   if (score >= 20)
  {
      break;
  }

   //had se pohne o jedno pole nasatvenym smerem
  had.snakeMove();
  //pokud jsou vsechny kousky jidla sesbirane ukonci simulaci

  //pokud narazi do zdi popr casti sveho tela ukonci simulaci
  if (! Collision(had))
  {
      break;
  }
  int smaz = 1;
  //testovani pokud zkonzumoval ovoce pokdu ano had poroste a dany kousek jidla se
      smaze
  for (int i = 0; i < jidlo.size(); i++)
  {
      if ( jidlo.get(i).posX == had.telo.get(0).posX && jidlo.get(i).posY == had.telo.get
          (0).posY)
      {
          had.growSnake();
          smaz = i;
          score += 1;
      }
  }
  if (smaz == 0) jidlo.remove(smaz);
}
  //nastaveni ohodnoceni
      int hits = score;
      //Fitness ohodnoceni 0 - nejlepsi 1 - nejhorsi
      Double sum = 1 - (double)score/20;

      KozaFitness f = ((KozaFitness)ind.fitness);
      f.setStandardizedFitness(state, sum);
      f.hits = hits;
      ind.evaluated = true;
      }
  }
```

Výpis 21: Třída SnakeProblem simulující hru Had.

4.6 Nastavení parametrů pro výpočet

Pomocí různých nastavení souboru s parametry lze docílit různých výsledku při spuštění genetického výpočtu. Lze změnit, jaké funkce budou a nebudou použity pro tvorbu Stromů, počet generací a jedinců v dané generaci. Všechny tyto nastavení mají vliv nato jestli a jak rychle najdeme jedince pro řešení našeho problému. Nastavení se odvíjí podle toho jak rychle chceme výsledek a jak výsledný nalezený jedinec má být kvalitní. V případě této hry je nastavení 50 generací o 1000 jedincích. K ideálnímu počtu jedinců v dané generaci jsme došli pomocí opakovaného testování. I v případě takového množství generací a jedinců je možné, že ECJ nebude schopno najít řešení našeho problému. Obvykle ECJ řešení pro náš problém našlo zhruba ve dvacáté generaci, ovšem občas nastane situace kdy se pří výpočtu ECJ zastaví na určité úrovní Fitness, přes kterou se dál není schopné dostat a v dalších generacích se pořád opakuje to samé ohodnocení beze změny. V tomto případě je nutné aplikaci spustit znovu. Základní nastavení použité pro řešení problému *SnakeProblem* 22. Tento výčet obsahuje jen pár nejdůležitějších nastavení.Toto nastavení lze nalézt v souboru *BP_1_params.params*.

5 Závěr

Cílem práce bylo popsat evoluční techniky a jejich využití při tvorbě počítačových programů a následná implementace těchto technik do vlastní aplikace. Pro využití Genetického programování byla vybrána hra Snake ve které algoritmus vygenerovaný pomocí ECJ naviguje hada po herní ploše. Původně měla hra obsahovat i implementaci překážek. Jelikož se ale zahrnutím překážek do simulace zásadně zvyšuje časovou náročnost na výpočet, nebyly nakonec překážky přidány.

V první kapitole byly popsány techniky Genetického algoritmu a Genetického programování.

Druhá kapitola se věnuje důkladnému popisu ECJ a jeho klíčových funkcí evolučních výpočtů Genetického algoritmu a Genetického programování. V této kapitole byl kladen důraz primárně na Genetické programování jeho podrobný analýza a popis technik které byly využity při tvorbě vlastní aplikace.

Třetí kapitola obsahuje popis a dokumentaci hry Snake implementované v programovacím jazyce Java využívajícího umělou inteligenci generovanou pomocí Genetického programováni systému ECJ. První část se věnovala grafickému rozhraní a popisu funkcionality aplikace. Druhá část této kapitoly se zaměřuje na konkrétní implementaci tříd využitých při tvorbě algoritmu ovládající umělou inteligenci ve hře.

Michal Koudela

6 Reference

[1] LUKE, Sean. The ECJ Owner's Manual [online]. 22. vyd., 2014 [cit. 2015-4-14]. Dostupné z: https://cs.gmu.edu/~eclab/projects/ecj/docs/manual/manual.pdf

[2] KOZA, John R. Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems. Stanford, c1990.

[3] KOZA, John R. Genetic programming: on the programming of computers by means of natural selection. Cambridge, Mass.: MIT Press, c1992, xiv, 819 p. ISBN 0262111705.

[4] MITCHELL, Melanie. An introduction to genetic algorithms. Cambridge: Bradford Book, c1996, viii, 205 s. ISBN 0262133164.

[5] EHLIS, Tobin. Application of Genetic Programming to the Snake Game. [online]. 2000 [cit. 2015-4-14]. Dostupné z: http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/application-of-genetic-programming-to-the-snake-r1175

7 Přílohy

7.1 Ukázka konfiguračního souboru

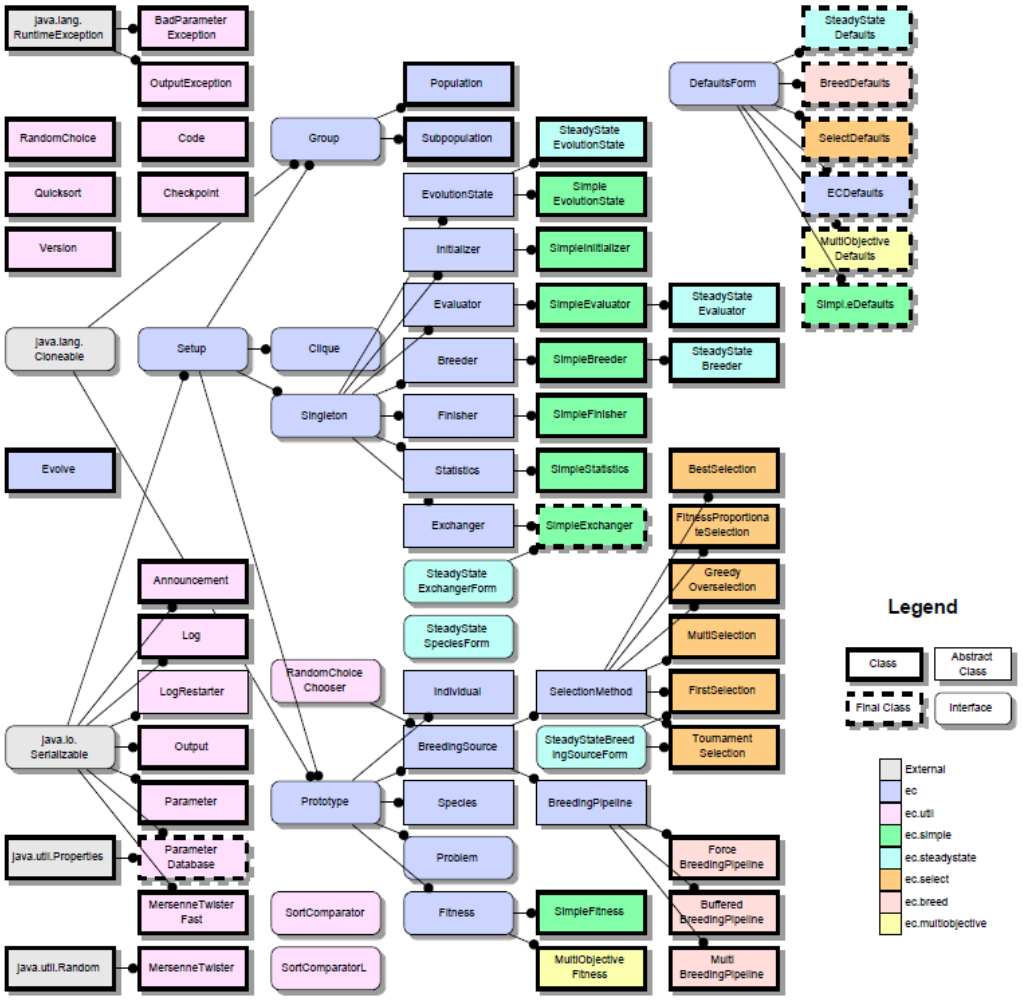
```
// nastaveni poctu generaci - 50
generations =               50
// pokud najde idealniho jedince jiz v 0generaci ukonci vypocet
quit-on-run-complete =      true
// velikost jedne populace cita 1000 jedincu
pop.subpop.0.size =      1000
// nazev souboru do ktere se bude ukladat vystup ECJ
stat. file                  $out.stat
// pocet funkci slouzicich ke konstrukci stromu
gp.fs.0.size = 14
// prirazeni funkci do sady Funkci a jejich nastaveni
gp.fs.0.func.0 = ec.app.myapp.Forward
gp.fs.0.func.0.nc = nc0
gp.fs.0.func.1 = ec.app.myapp.Left
gp.fs.0.func.1.nc = nc0
gp.fs.0.func.2 = ec.app.myapp.Right
gp.fs.0.func.2.nc = nc0
gp.fs.0.func.3 = ec.app.myapp.ifDangerAhead
gp.fs.0.func.3.nc = nc2
gp.fs.0.func.4 = ec.app.myapp.ifFoodUp
gp.fs.0.func.4.nc = nc2
gp.fs.0.func.5 = ec.app.myapp.ifFoodRight
gp.fs.0.func.5.nc = nc2
gp.fs.0.func.6 = ec.app.myapp.ifMovingLeft
gp.fs.0.func.6.nc = nc2
gp.fs.0.func.7 = ec.app.myapp.ifMovingRight
gp.fs.0.func.7.nc = nc2
gp.fs.0.func.8 = ec.app.myapp.ifDangerTwoAhead
gp.fs.0.func.8.nc = nc2
gp.fs.0.func.9 = ec.app.myapp.ifMovingUp
gp.fs.0.func.9.nc = nc2
gp.fs.0.func.10 = ec.app.myapp.ifMovingDown
gp.fs.0.func.10.nc = nc2
gp.fs.0.func.11 = ec.app.myapp.ifDangerLeft
gp.fs.0.func.11.nc = nc2
gp.fs.0.func.12 = ec.app.myapp.ifDangerRight
gp.fs.0.func.12.nc = nc2
gp.fs.0.func.13 = ec.app.myapp.ifFoodAhead
gp.fs.0.func.13.nc = nc2

// prirazeni tridy nesouci data pro vypocet ktere se predavaji mezi
prvky stromu a reseny problem SnakeProblem
eval.problem.data = ec.app.myapp.Direction
eval.problem = ec.app.myapp.SnakeProblem
```

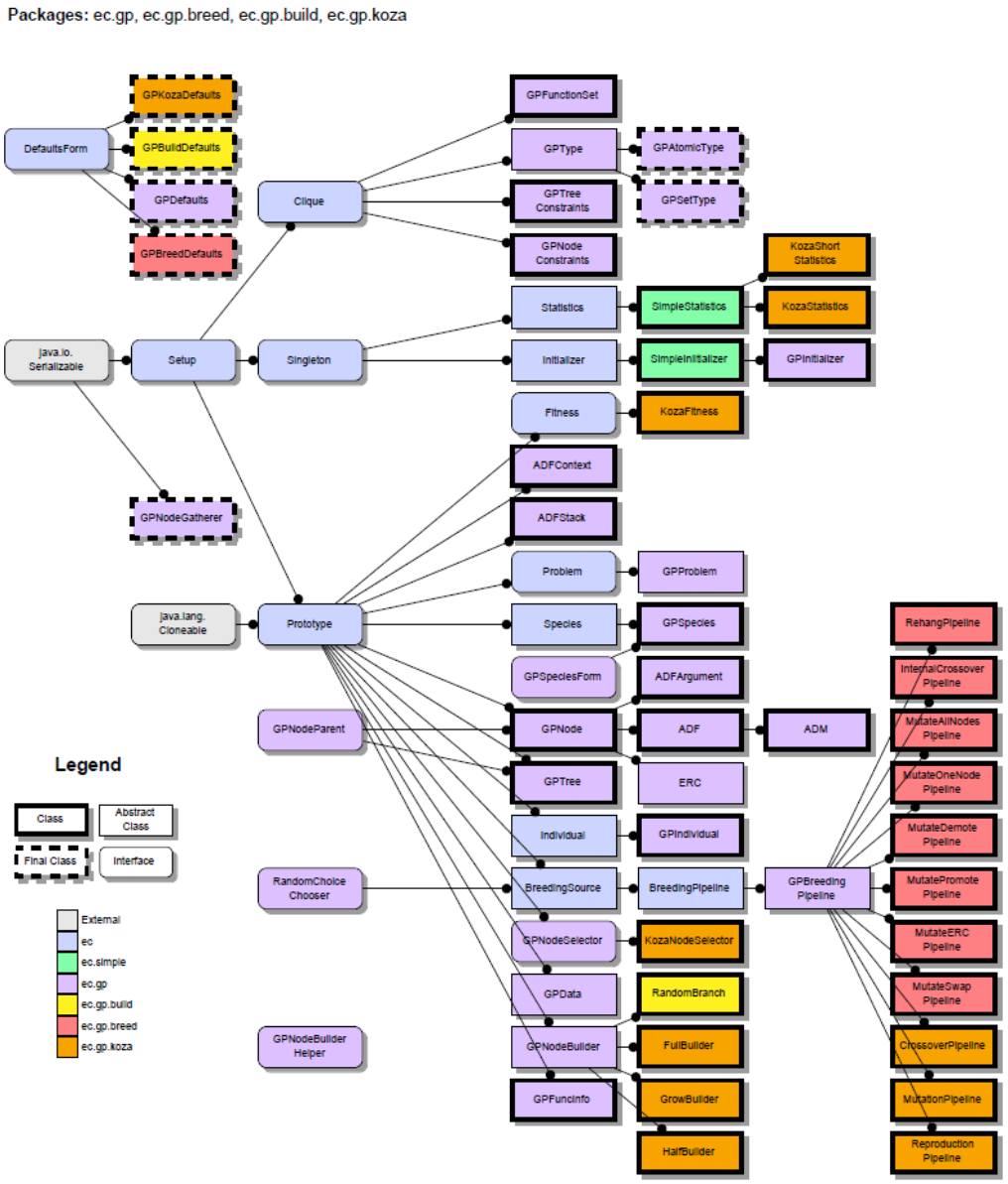
Výpis 22: Ukázka části konfiguračního souboru.

7.2 Struktura tříd v ECJ

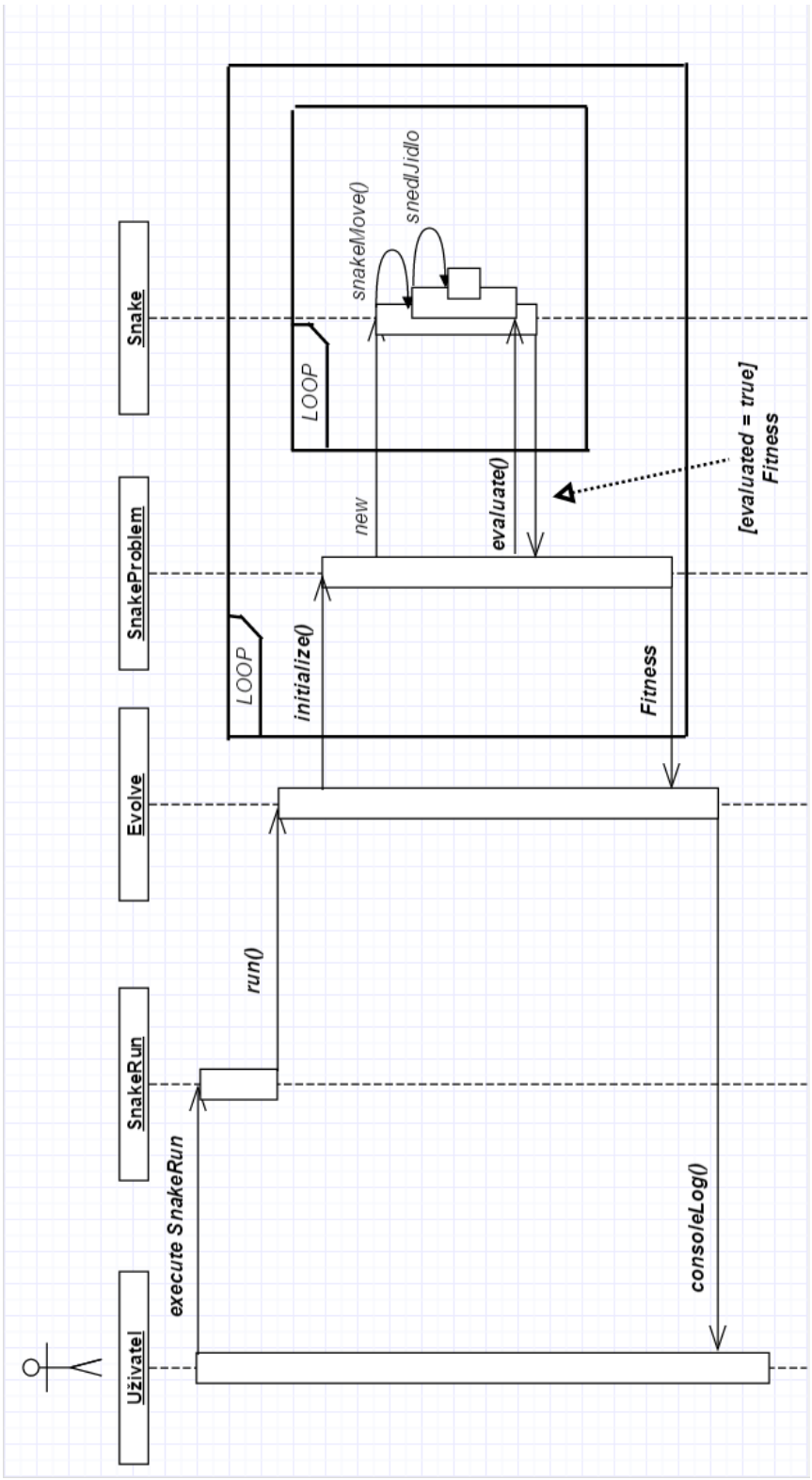
Packages: ec, ec.util, ec.simple, ec.steadystate, ec.breed, ec.multiobjective



Obrázek 9: Struktura tříd v ECJ [1]



Obrázek 10: Struktura tříd v ECJ část 2 [1]



Obrázek 11: Sekvenční diagram zobrazující průběh výpočtu ECJ ve hře Snake